

Document Retrieval on Repetitive Collections[★]

Gonzalo Navarro¹, Simon J. Puglisi², and Jouni Sirén¹

¹ Center for Biotechnology and Bioengineering, Department of Computer Science,
University of Chile, Chile

`{gnavarro, jsiren}@dcc.uchile.cl`

² Department of Computer Science, University of Helsinki, Finland
`puglisi@cs.helsinki.fi`

Abstract. Document retrieval aims at finding the most important documents where a pattern appears in a collection of strings. Traditional pattern-matching techniques yield brute-force document retrieval solutions, which has motivated the research on tailored indexes that offer near-optimal performance. However, an experimental study establishing which alternatives are actually better than brute force, and which perform best depending on the collection characteristics, has not been carried out. In this paper we address this shortcoming by exploring the relationship between the nature of the underlying collection and the performance of current methods. Via extensive experiments we show that established solutions are often beaten in practice by brute-force alternatives. We also design new methods that offer superior time/space trade-offs, particularly on repetitive collections.

1 Introduction

The *pattern matching* problem, that is, preprocessing a text collection so as to efficiently find the occurrences of patterns, is a classic in Computer Science. The optimal suffix tree solution [18] dates back to 1973. Suffix arrays [10] are a simpler, near-optimal alternative. Surprisingly, the natural variant of the problem called *document listing*, where one wants to find simply in which texts of the collection (called the *documents*) a pattern appears, was not solved optimally until almost 30 years later [11]. Another natural variant, the *top-k documents* problem, where one wants to find the *k most relevant* documents where a pattern appears, for some notion of relevance, had to wait for other 10 years [6,15].

A general problem with the above indexes is their size. While for moderate-sized collections (of total length n) their linear space (i.e., $O(n)$ words, or $O(n \log n)$ bits) is affordable, the constant factors multiplying the linear term make the solutions prohibitive on large collections. In this aspect, again, the pattern matching problem has had some years of advantage. The first compressed

[★] This work is funded in part by: Fondecyt Project 1-140796 (first author); Basal Funds FB0001, Conicyt, Chile (first and third authors); the Jenny and Antti Wihuri Foundation, Finland (third author); and by the Academy of Finland through grants 258308 and 250345 (CoECGR) (second author).

suffix arrays (CSAs) appeared in the year 2000 (see [14]) and since then have evolved until achieving, for example, asymptotically optimal space in terms of high-order empirical entropy and time slightly over the optimal. There has been much research on similarly compressed data structures for document retrieval (see [13]). Since the foundational paper of Hon et al. [6], results have come close to using just $o(n)$ bits on top of the space of a CSA and almost optimal time.

Compressing in terms of statistical entropy is adequate in many cases, but it fails in various types of modern collections. *Repetitive* document collections, where most documents are similar, in whole or piecewise, to other documents, naturally arise in fields like computational biology, versioned collections, periodic publications, and software repositories (see [12]). The successful pattern matching indices for these types of collections use grammar or Lempel-Ziv compression, which exploit repetitiveness [2,3]. There are only a couple of document listing indices for repetitive collections [4,1], and none for the top- k problem.

Although several document retrieval solutions have been implemented and tested in practice [16,7,3,4], no systematic practical study of how these indexes perform, depending on the collection characteristics, has been carried out.

A first issue is to determine under what circumstances specific document listing solutions actually beat brute-force solutions based on pattern matching. In many applications documents are relatively small (a few kilobytes) and therefore are unlikely to contain many occurrences of a given pattern. This means that in practice the number of pattern occurrences (*occ*) may not be much larger than the number of documents the pattern occurs in (*docc*), and therefore pattern matching-based solutions may be competitive.

A second issue that has been generally neglected in the literature is that collections have different kinds of repetitiveness, depending on the application. For example, one might have a set of distinct documents, each one internally repetitive piecewise, or a set of documents that are in whole similar to each other. The repetition structure can be linear (each document similar to a previous one) as in versioned collections, or even tree-like, or completely unstructured, as in some biological collections. It is not clear how current document retrieval solutions behave depending on the type of repetitiveness.

In this paper we carry out a thorough experimental study of the performance of most existing solutions to document listing and top- k document retrieval, considering various types of real-life and synthetic collections. We show that brute-force solutions are indeed competitive in several practical scenarios, and that some existing solutions perform well only on some kinds of repetitive collections, whereas others present a more stable behavior. We also design new and superior alternatives for top- k document retrieval.

2 Background

Let $T[1, n]$ be a concatenation of a collection of d documents. We assume each document ends with a special character $\$$ that is lexicographically smaller than any other character of the alphabet. The *suffix array* of the collection is an array

$SA[1, n]$ of pointers to the suffixes of T in lexicographic order. The *document array* $DA[1, n]$ is a related array, where $DA[i]$ is the identifier of the document containing $T[SA[i]]$. Let $B[1, n]$ be a bitvector, where $B[i] = 1$ if a new document begins at $T[i]$. We can map text positions to document identifiers by: $DA[i] = \text{rank}_1(B, SA[i])$, where $\text{rank}_1(B, j)$ is the number of 1-bits in prefix $B[1, j]$.

In this paper, we consider indexes supporting four kinds of queries: 1) $\text{find}(P)$ returns the range $[sp, ep]$, where the suffixes in $SA[sp, ep]$ start with pattern P ; 2) $\text{locate}(sp, ep)$ returns $SA[sp, ep]$; 3) $\text{list}(P)$ returns the identifiers of documents containing pattern P ; and 4) $\text{topk}(P, k)$ returns the identifiers of the k documents containing the most occurrences of P . CSAs support the first two queries. $\text{find}()$ is relatively fast, while $\text{locate}()$ can be much slower. The main time/space trade-off in a CSA, the *suffix array sample period*, affects the performance of $\text{locate}()$ queries. Larger sample periods result in slower and smaller indexes.

Muthukrishnan’s document listing algorithm [11] uses an array $C[1, n]$, where $C[i]$ points to the last occurrence of $DA[i]$ in $DA[1, i - 1]$. Given a query range $[sp, ep]$, $DA[i]$ is the first occurrence of that document in the range iff $C[i] < sp$. A *range minimum query* (RMQ) structure over C is used to find the position i with the smallest value in $C[sp, ep]$. If $C[i] < sp$, the algorithm reports $DA[i]$, and continues recursively in $[sp, i - 1]$ and $[i + 1, ep]$. Sadakane [17] improved the space usage with two observations: 1) if the recursion is done in preorder from left to right, $C[i] \geq sp$ iff document $DA[i]$ has been seen before, so array C is not needed; and 2) array DA can also be removed by using $\text{locate}()$ and B instead.

Let $\text{lcp}(S, T)$ be the length of the *longest common prefix* of sequences S and T . The LCP array of $T[1, n]$ is an array $\text{LCP}[1, n]$, where $\text{LCP}[i] = \text{lcp}(T[SA[i - 1], n], T[SA[i], n])$. We obtain the *interleaved LCP array* $\text{ILCP}[1, n]$ by building separate LCP arrays for each of the documents, and interleaving them according to the document array. As $\text{ILCP}[i] < |P|$ iff position i contains the first occurrence of $DA[i]$ in $DA[sp, ep]$, we can use Sadakane’s algorithm with RMQs over ILCP instead of C [4]. If the collection is repetitive, we can get a smaller and faster index by building the RMQ only over the run heads in ILCP .

3 Algorithms

In this section we review *practical* methods for document listing and top- k document retrieval. For a more detailed review see, e.g., [13].

Brute force. These algorithms sort the document identifiers in range $DA[sp, ep]$ and report each of them once. **Brute-D** stores DA in $n \log d$ bits, while **Brute-L** retrieves the range $SA[sp, ep]$ with $\text{locate}()$ and uses bitvector B to convert it to $DA[sp, ep]$. Both algorithms can also be used for top- k retrieval by computing the frequency of each document identifier and then sorting by frequency.

Sadakane. This is a family of algorithms based on Sadakane’s improvements [17] to Muthukrishnan’s algorithm [11]. **Sada-C-L** is the original algorithm of Sadakane, while **Sada-C-D** uses an explicit document array instead of retrieving the document identifiers with $\text{locate}()$. **Sada-I-L** and **Sada-I-D** are otherwise the same, respectively, except that they build the RMQ over ILCP [4] instead of C .

Wavelet tree. A *wavelet tree* over a sequence can be used to quickly list the distinct values in any substring, and hence a wavelet tree over DA can be a good solution for many document retrieval problems. The best known implementation of wavelet tree-based document listing [16] can use plain, entropy-compressed [14], and grammar-compressed [8] bitvectors in the wavelet tree. Our WT uses a heuristic similar to the original WT-alpha [16], multiplying the size of the plain bitvector by 0.81 and the size of the entropy-compressed bitvector by 0.9, before choosing the smallest one for each level of the tree.

For top- k retrieval, WT combines the wavelet tree used in document listing with a space-efficient implementation [16] of the top- k trees of Hon et al. [6]. Out of the alternatives investigated by Navarro and Valenzuela [16], we tested the greedy algorithm, LIGHT and XLIGHT encodings for the trees, and sampling parameter $g' = 400$. In the results, we use the slightly smaller XLIGHT.

Precomputed document listing. PDL [4] builds a sparse suffix tree for the collection, and stores the answers to document listing queries for the nodes of the tree. For long query ranges, we compute the answer to the `list()` query as a union of a small number of stored answer sets. The answers for short ranges are computed by using Brute-L. PDL-BC is the original version, using a web graph compressor [5] to compress the sets. If a subset S' of document identifiers occurs in many of the stored sets, the compressor creates a grammar rule $X \rightarrow S'$, and replaces the subset with X . We chose block size $b = 256$ and storing factor $\beta = 16$ as good general-purpose parameter values. We extend PDL in Section 4.

Grammar-based. Grammar [1] is an adaptation of a grammar-compressed self-index [2] for document listing. Conceptually similar to PDL, Grammar uses Re-Pair [8] to parse the collection. For each nonterminal symbol in the grammar, it stores the set of document identifiers whose encoding contains the symbol. A second round of Re-Pair is used to compress the sets. Unlike most of the other solutions, Grammar is an independent index and needs no CSA to operate.

Lempel-Ziv. LZ [3] is an adaptation of self-indexes based on LZ78 parsing for document listing. Like Grammar, LZ does not need a CSA.

Grid. Grid [7] is a faster but usually larger alternative to WT. It can answer top- k queries quickly if the pattern occurs at least twice in each reported document. If documents with just one occurrence are needed, Grid uses a variant of Sada-C-L to find them. We also tried to use Grid for document listing, but the performance was not good, as it usually reverted to Sada-C-L.

4 Extending Precomputed Document Listing

In addition to PDL-BC, we implemented another variant of precomputed document listing [4] that uses Re-Pair [8] instead of the biclique-based compressor.

In the new variant, named PDL-RP, each stored set is represented as an increasing sequence of document identifiers. The stored sets are compressed with Re-Pair, but otherwise PDL-RP is the same as PDL-BC. Due to the multi-level grammar generated by Re-Pair, decompressing the sets can be slower in PDL-RP than in PDL-BC. Another drawback comes from representing the sets as

sequences: when the collection is non-repetitive, Re-Pair cannot compress the sets very well. On the positive side, compression is much faster and more stable.

We also tried an intermediate variant, PDL-set, that uses Re-Pair-like set compression. While ordinary Re-Pair replaces common substrings ab of length 2 with grammar rules $X \rightarrow ab$, the compressor used in PDL-set searches for symbols a and b that occur often in the same sets. Treating the sets this way should lead to better compression on non-repetitive collections, but unfortunately our current compression algorithm is still too slow with non-repetitive collections. With repetitive collections, the size of PDL-set is very similar to PDL-RP.

Representing the sets as sequences allows for storing the document identifiers in any desired order. One interesting order is the top- k order: store the identifiers in the order they should be returned by a `topk()` query. This forms the basis of our new PDL structure for top- k document retrieval. In each set, document identifiers are sorted by their frequencies in decreasing order, with ties broken by sorting the identifiers in increasing order. The sequences are then compressed by Re-Pair. If document frequencies are needed, they are stored in the same order as the identifiers. The frequencies can be represented space-efficiently by first run-length encoding the sequences, and then using differential encoding for the run heads. If there are b suffixes in the subtree corresponding to the set, there are $O(\sqrt{b})$ runs, so the frequencies can be encoded in $O(\sqrt{b} \log b)$ bits.

There are two basic approaches to using the PDL structure for top- k document retrieval. We can set $\beta = 0$, storing the document sets for all suffix tree nodes above the leaf blocks. This approach is very fast, as we need only decompress the first k document identifiers from the stored sequence. It works well with repetitive collections, while the total size of the document sets becomes too large with non-repetitive collections. We tried this approach with block sizes $b = 64$ (PDL-64 without frequencies and PDL-64+F with frequencies) and $b = 256$ (PDL-256 and PDL-256+F).

Alternatively, we can build the PDL structure normally with $\beta > 1$, achieving better compression. Answering queries is now slower, as we have to decompress multiple document sets with frequencies, merge the sets, and determine the top k . We tried different heuristics for merging only prefixes of the document sequences, stopping when a correct answer to the top- k query could be guaranteed. The heuristics did not generally work well, making brute-force merging the fastest alternative. We used block size $b = 256$ and storing factors $\beta = 2$ (PDL-256-2) and $\beta = 4$ (PDL-256-4). Smaller block sizes increased both index size and query times, as the number of sets to be merged was generally larger.

5 Experimental Data

We did extensive experiments with both real and synthetic collections.³ The details of the collections can be seen in Table 1 in the Appendix, where we also describe how the search patterns were obtained.

³ See <http://www.cs.helsinki.fi/group/suds/rlcsa/> for datasets and full results.

Most of our document collections were relatively small, around 100 MB in size, as the WT implementation uses 32-bit libraries, while Grid requires large amounts of memory for index construction. We also used larger versions of some collections, up to 1 GB in size, to see how collection size affects the results. In practice, collection size was more important in top- k document retrieval, as increasing the number of documents generally increases the $docc/k$ ratio. In document listing, document size is more important than collection size, as the performance of Brute depends on the $occ/docc$ ratio.

Real collections. Page and Revision are repetitive collections generated from a Finnish language Wikipedia archive with full version history. The collection consists of either 60 pages (small) or 280 pages (large), with a total of 8834 or 65565 revisions. In Page, all revisions of a page form a single document, while each revision becomes a separate document in Revision. Enwiki is a nonrepetitive collection of 7000 or 90000 pages from a snapshot of the English language Wikipedia. Influenza is a repetitive collection containing the genomes of 100000 or 227356 influenza viruses. Swissprot is a nonrepetitive collection of 143244 protein sequences used in many document retrieval papers (e.g., [16]). As the full collection is only 54 MB, there is no large version of Swissprot.

Synthetic collections. To explore the effect of collection repetitiveness on document retrieval performance in more detail, we generated three types of synthetic collections, using files from the Pizza & Chilli corpus⁴.

DNA is similar to Influenza. Each collection has 1, 10, 100, or 1000 base documents, 100000, 10000, 1000, or 100 variants of each base document, respectively, and mutation rate $p = 0.001, 0.003, 0.01, 0.03$, or 0.1 . We generated the base documents by mutating a sequence of length 1000 from the DNA file with zero-order entropy preserving point mutations, with probability $10p$. We then generated the variants in the same way with mutation rate p .

Concat is similar to Page. We read 10, 100, or 1000 base documents of length 10000 from the English file, and generated 1000, 100, or 10 variants of each base document, respectively. The variants were generated by applying zero-order entropy preserving point mutations with probability $0.001, 0.003, 0.01, 0.03$, or 0.1 to the base document, and all variants of a base document were concatenated to form a single document. We also generated collections similar to Revision by making each variant a separate document. These collections are called Version.

6 Experimental Results

We implemented Brute, Sada, and PDL ourselves⁵, and modified existing implementations of WT, Grid, Grammar, and LZ for our purposes. All implementations were written in C++. Details of our test machine are in the Appendix.

As our CSA, we used RLCSA [9], a practical implementation of a CSA that compresses repetitive collections well. The `locate()` support in RLCSA includes optimizations for long query ranges and repetitive collections, which is important

⁴ <http://pizzachili.dcc.uchile.cl/>

⁵ Available at <http://www.cs.helsinki.fi/group/suds/rlcsa/>

for Brute-L and Sada-I-L. We used suffix array sample periods 8, 16, 32, 64, 128 for non-repetitive collections and 32, 64, 128, 256, 512 for repetitive ones.

For algorithms using a CSA, we broke the $\text{list}(P)$ and $\text{topk}(P, k)$ queries into a $\text{find}(P)$ query, followed by a $\text{list}([sp, ep])$ query or $\text{topk}([sp, ep], k)$ query, respectively. The measured times do not include the time used by the $\text{find}()$ query. As this time is common to all solutions using a CSA, and negligible compared to the time used by Grammar and LZ, the omission does not affect the results.

Document listing with real collections. Figure 1 contains the results for document listing with real collections. For most of the indexes, the time/space trade-off is based on the SA sample period. LZ’s trade-off comes from a parameter specific to that structure involving RMQs (see [3]). Grammar has no trade-off.

Of the small indexes, Brute-L is usually the best choice. Thanks to the $\text{locate}()$ optimizations in RLCSA and the small documents, Brute-L beats Sada-C-L and Sada-I-L, which are faster in theory due to using $\text{locate}()$ more selectively. When more space is available, PDL-BC is a good choice, combining fast queries with moderate space usage. Of the bigger indexes, one storing the document array explicitly is usually even faster than PDL-BC. Grammar works well with Revision and Influenza, but becomes too large or too slow elsewhere.

Top- k document retrieval. Results for top- k document retrieval on real collections are shown in Figures 2 and 3. Time/space trade-offs are again based on the suffix array sample period, while PDL also uses other parameters (see Section 4). We could not build PDL with $\beta = 0$ for Influenza or the large collections, as the total size of the stored sets was more than 2^{32} , which was too much for our Re-Pair compressor. WT was only built for the small collections, while Grid construction used too much memory on the larger Wikipedia collections.

On Revision, PDL dominates the other solutions. On Enwiki, both WT and Grid have good trade-offs with $k = 10$, while Brute-D and PDL beat them with $k = 100$. On Influenza, some PDL variants, Brute-D, and Grid all offer good trade-offs. On Swissprot, the brute-force algorithms win clearly. PDL with $\beta = 0$ is faster, but requires far too much space (60-70 bpc — off the chart).

Document listing with synthetic collections. Figure 4 shows our document listing results with synthetic collections. Due to the large number of collections, the results for a given collection type and number of base documents are combined in a single plot, showing the fastest algorithm for a given amount of space and a mutation rate. Solid lines connect measurements that are the fastest for their size, while dashed lines are rough interpolations.

The plots were simplified in two ways. Algorithms providing a marginal and/or inconsistent improvement in speed in a very narrow region (mainly Sada-C-L and Sada-I-L) were left out. When PDL-BC and PDL-RP had very similar performance, only one of them was chosen for the plot.

On DNA, Grammar was a good solution for small mutation rates, while LZ was good with larger mutation rates. With more space available, PDL-BC became the fastest algorithm. Brute-D and Sada-I-D were often slightly faster than PDL, when there was enough space available to store the document array. On Concat and Version, PDL was usually a good mid-range solution, with PDL-RP being

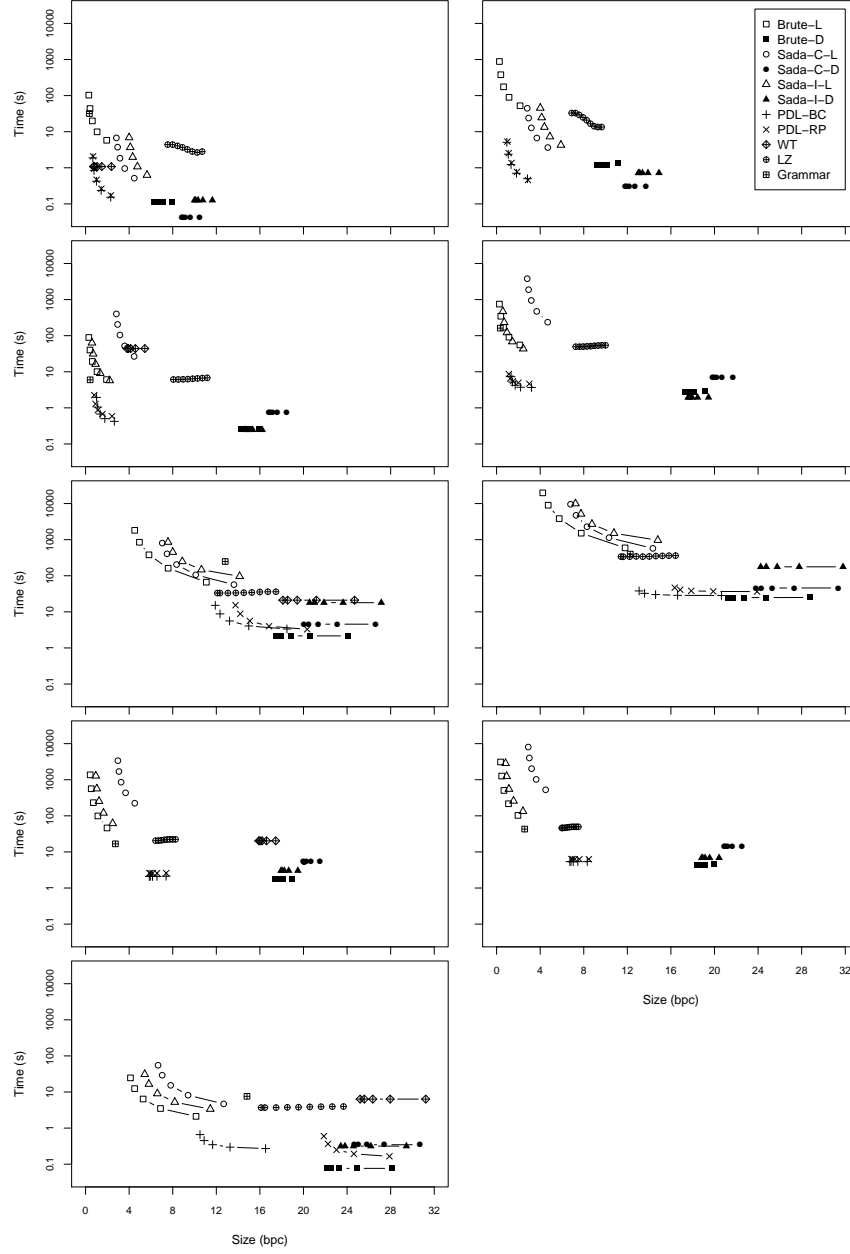


Fig. 1. Document listing on small (left) and large (right) real collections. Total size of the index in bits per character and time required to run the queries in seconds. From top to bottom, Page, Revision, Enwiki, Influenza, and Swissprot.

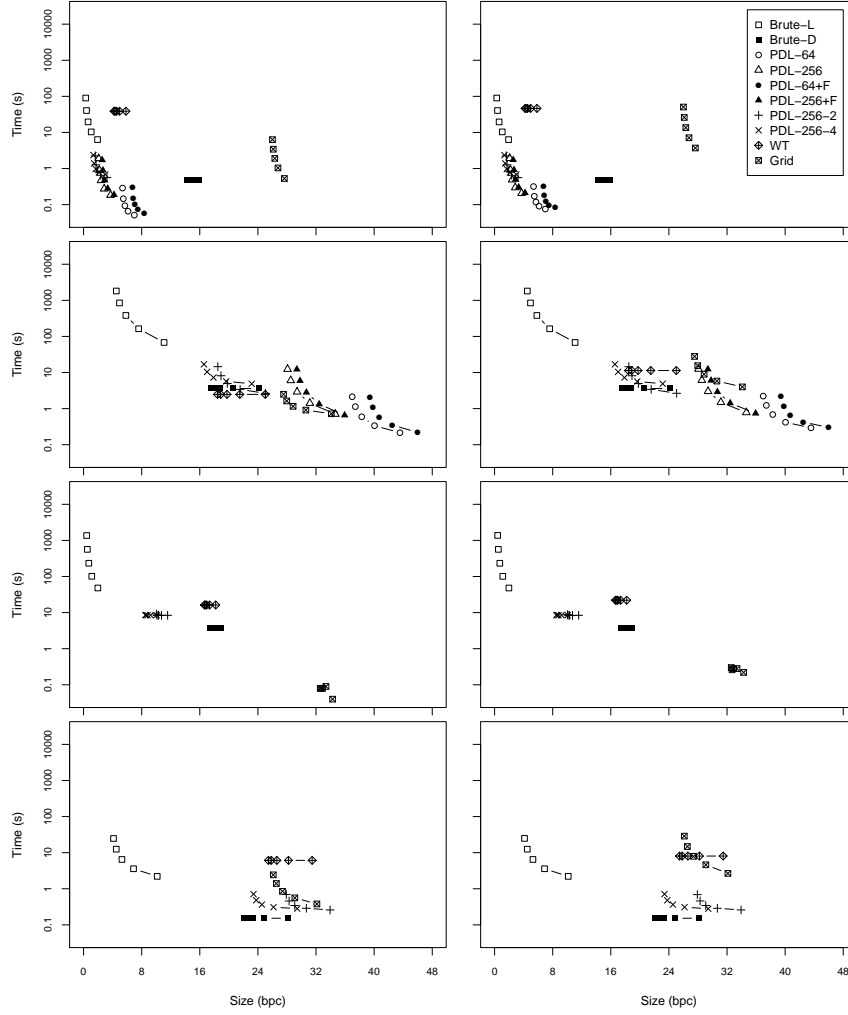


Fig. 2. Top- k document retrieval with $k = 10$ (left) and $k = 100$ (right) on small real collections. Total size of the index in bits per character and time required to run the queries in seconds. From top to bottom, Revision, Enwiki, Influenza, and Swissprot. Page is left out due to the low number of documents in that collection.

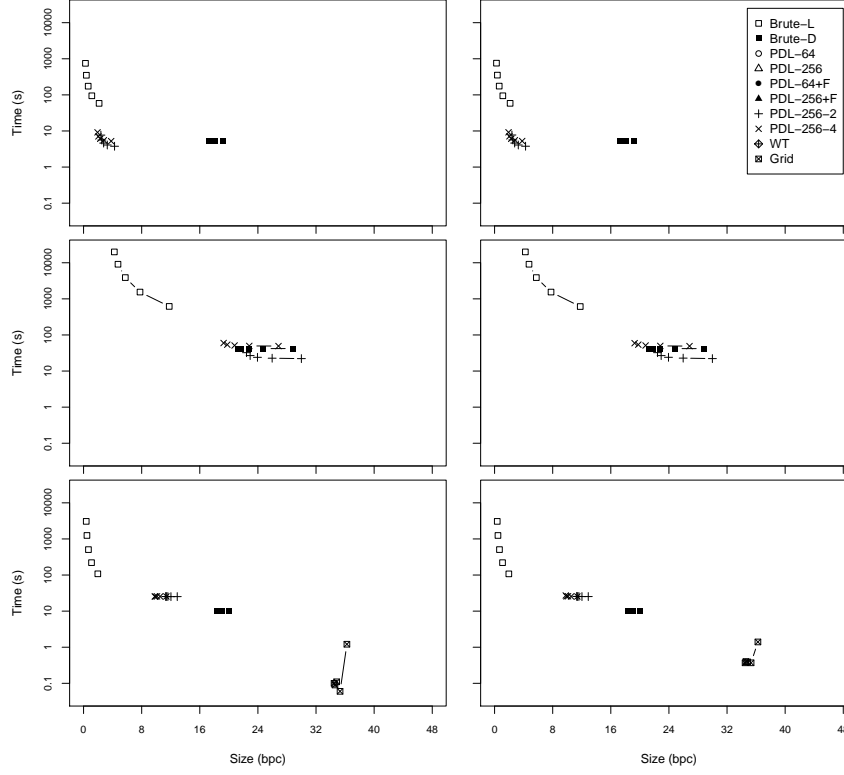


Fig. 3. Top- k document retrieval with $k = 10$ (left) and $k = 100$ (right) on large real collections. Total size of the index in bits per character and time required to run the queries in seconds. From top to bottom, Revision, Enwiki, and Influenza. Page is left out due to the low number of documents in that collection.

usually smaller than PDL-BC. The exceptions were the collections with 10 base documents, where the number of variants (1000) was clearly larger than the block size (256). With no other structure in the collection, PDL was unable to find a good grammar to compress the sets. At the large end of the size scale, algorithms using an explicit DA were usually the fastest choice.

7 Conclusions

Most document listing algorithms assume that the total number of occurrences of the pattern is large compared to the number of document occurrences. When documents are small, such as Wikipedia articles, this assumption generally does not hold. In such cases, brute-force algorithms usually beat dedicated document listing algorithms, such as Sadakane’s algorithm and wavelet tree-based ones.

Several new algorithms have been proposed recently. PDL is a fast and small solution, effective on non-repetitive collections, and with repetitive collections, if

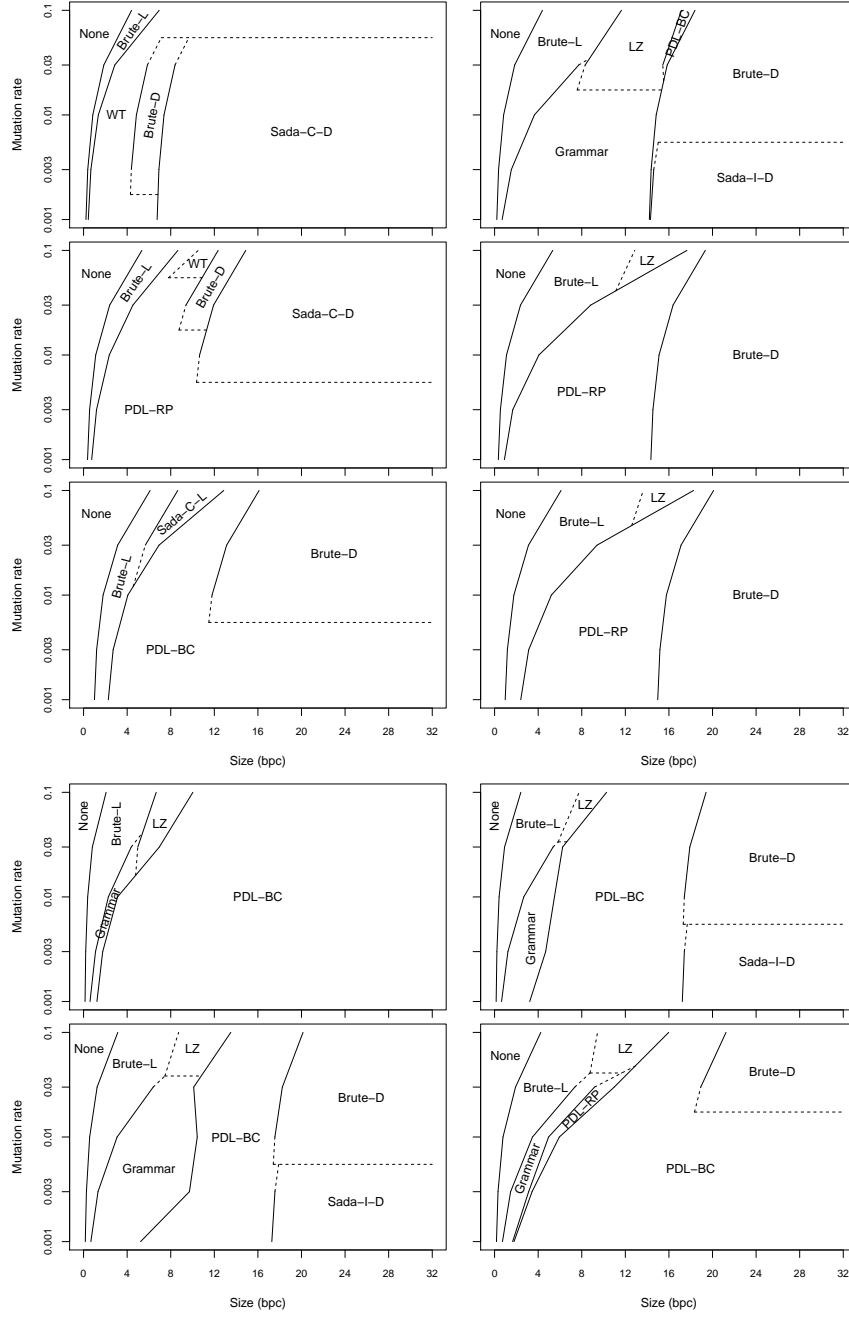


Fig. 4. Document listing with synthetic collections. The fastest solution for a given size in bits per character and a mutation rate. Top group: from top to bottom 10, 100, and 1000 base documents with **Concat** (left) and **Version** (right). Bottom group: DNA with 1 (top left), 10 (top right), 100 (bottom left), and 1000 (bottom right) base documents. **None** denotes that no solution can achieve that size.

the collection is structured (e.g., incremental versions of base documents) or the average number of similar suffixes is not too large. Of the two PDL variants, PDL-BC has a more stable performance, while PDL-RP is faster to build. Grammar is a small and moderately fast solution when the collection is repetitive but the individual documents are not. LZ works well when repetition is moderate.

We adapted the PDL structure for top- k document retrieval. The new structure works well with repetitive collections, and is clearly the method of choice on the versioned Revision. When the collections are non-repetitive, brute-force algorithms remain competitive even on gigabyte-sized collections. While some dedicated algorithms can be faster, the price is much higher space usage.

References

1. F. Claude and I. Munro. Document listing on versioned documents. In *Proc. SPIRE*, LNCS 8214, pages 72–83, 2013.
2. F. Claude and G. Navarro. Improved grammar-based compressed indexes. In *Proc. SPIRE*, LNCS 7608, pages 180–192, 2012.
3. H. Ferrada and G. Navarro. A Lempel-Ziv compressed structure for document listing. In *Proc. SPIRE*, LNCS 8214, pages 116–128, 2013.
4. T. Gagie, K. Karhu, G. Navarro, S. J. Puglisi, and J. Sirén. Document listing on repetitive collections. In *Proc. CPM*, LNCS 7922, pages 107–119, 2013.
5. C. Hernández and G. Navarro. Compressed representation of web and social networks via dense subgraphs. In *Proc. SPIRE*, LNCS 7608, pages 264–276, 2012.
6. W.-K. Hon, R. Shah, and J. Vitter. Space-efficient framework for top- k string retrieval problems. In *Proc. FOCS*, pages 713–722, 2009.
7. R. Konow and G. Navarro. Faster compact top- k document retrieval. In *Proc. DCC*, pages 351–360, 2013.
8. N. J. Larsson and A. Moffat. Off-line dictionary-based compression. *Proceedings of the IEEE Data Compression Conference*, 88(11):1722–1732, 2000.
9. V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki. Storage and retrieval of highly repetitive sequence collections. *J. Comp. Bio.*, 17(3):281–308, 2010.
10. U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. Computing*, 22(5):935–948, 1993.
11. S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proc. SODA*, pages 657–666, 2002.
12. G. Navarro. Indexing highly repetitive collections. In *Proc. IWOCOA*, LNCS 7643, pages 274–279, 2012.
13. G. Navarro. Spaces, trees and colors: The algorithmic landscape of document retrieval on sequences. *ACM Computing Surveys*, 46(4):article 52, 2014.
14. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):art. 2, 2007.
15. G. Navarro and Y. Nekrich. Top- k document retrieval in optimal time and linear space. In *Proc. SODA*, pages 1066–1078, 2012.
16. G. Navarro and D. Valenzuela. Space-efficient top- k document retrieval. In *Proc. SEA*, LNCS 7276, pages 307–319, 2012.
17. K. Sadakane. Succinct data structures for flexible text retrieval systems. *J. Discrete Algorithms*, 5:12–22, 2007.
18. P. Weiner. Linear pattern matching algorithm. In *Proc. 14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.

Appendix

Test Environment

All implementations were written in C++ and compiled on g++ version 4.6.3. Our test environment was a machine with two 2.40 GHz quad-core Xeon E5620 processors (12 MB cache each) and 96 GB memory. Only one core was used for the queries. The operating system was Ubuntu 12.04 with Linux kernel 3.2.0.

Collections

Table 1. Statistics for document collections. Collection size, CSA size without suffix array samples, number of documents, average document length, number of patterns, average number of occurrences and document occurrences, and the ratio of occurrences to document occurrences. For synthetic collections, most of the statistics vary greatly.

Collection	Size	CSA	Documents	n/d	Patterns	\overline{occ}	\overline{docc}	$occ/docc$
Page	110 MB	2.58 MB	60	1919382	7658	781	3	242.75
	1037 MB	17.45 MB	280	3883145	20536	2889	7	429.04
Revision	110 MB	2.59 MB	8834	13005	7658	776	371	2.09
	1035 MB	17.55 MB	65565	16552	20536	2876	1188	2.42
Enwiki	113 MB	49.44 MB	7000	16932	18935	1904	505	3.77
	1034 MB	482.16 MB	90000	12050	19805	17092	4976	3.44
Influenza	137 MB	5.52 MB	100000	1436	1000	24975	18547	1.35
	321 MB	10.53 MB	227356	1480	1000	59997	44012	1.36
Swissprot	54 MB	25.19 MB	143244	398	10000	160	121	1.33
DNA	95 MB		100000		889–1000			
Concat	95 MB		10–1000		7538–15272			
Version	95 MB		10000		7537–15271			

Patterns

Real collections. For Page and Revision, we downloaded a list of Finnish words from the Institute for the Languages in Finland, and chose all words of length ≥ 5 that occur in the collection.

For Enwiki, we used search terms from an MSN query log with stop words filtered out. We generated 20000 patterns according to term frequencies, and selected those that occur in the collection.

For Influenza, we extracted 100000 random substrings of length 7, filtered out duplicates, and kept the 1000 patterns with the largest $occ/docc$ ratios.

For Swissprot, we extracted 200000 random substrings of length 5, filtered out duplicates, and kept the 10000 patterns with the largest $occ/docc$ ratios.

Synthetic collections. For DNA, patterns were generated with a similar process as for Influenza and Swissprot: take 100000 substrings of length 7, filter out duplicates, and choose the 1000 with the largest *occ/docc* ratios.

For Concat and Version, patterns were generated from the MSN query log in the same way as for Enwiki.